Performance Improvement by Using Pipelined Execution on Hyperledger Fabric

Ence Zhou
Fujitsu R&D Center Co., Ltd.
zhouence@fujitsu.com

Bingfeng Pi
Fujitsu R&D Center Co., Ltd.
winter.pi@fujitsu.com

Jun Sun
Fujitsu R&D Center Co., Ltd.
sunjun@fujitsu.com

Takeshi Miyamae
Fujitsu Limited
miyamae.takeshi@fujitsu.com

Masanobu Morinaga
Fujitsu Limited
morinaga@fujitsu.com

Abstract

The rapid growth of proofs of concept blockchain applications leads to increasing interest in understanding and improving blockchain performance at scale. However, the lower performance of blockchain restricts its application in some fields. Our work is focused on evaluating and improving the performance of Hyperledger Fabric, which is the most popular blockchain platform for enterprises. In previous works, the major bottleneck incurred in the validation & commit (V&C) module was studied, and many performance issues arising with it were alleviated to some context. The throughput is still only 900 transactions/second in our experiment. In this paper, a comprehensive latency evaluation for the V&C module was first performed. Then, according to the analysis of the evaluation results, a pipelined execution technology was proposed to process multiple blocks in parallel. Additionally, some pipeline acceleration schemes were also proposed to further improve the performance. Our experiments indicated performance improvements of 4.38× for LevelDB and at least 2× for CouchDB. Notably, our optimizations are transparent to blockchain applications and are suitable for integrating into a future version of Fabric.

1. Introduction

Blockchain technologies are becoming more and more popular due to their immutability of ledger data, and hence, it's easy to establish trust among untrusted participants. In a blockchain network, each node keeps an independent ledger, which is composed of a series of blocks. Each block contains a set of transactions and the hash value of the previous block, so as to protect the immutability of the ledger. Blockchain networks can be divided into permissionless ones and permissioned ones [1]. Anyone can submit transactions in a permissionless blockchain network, but needs to be authenticated before joining a permissioned one.

Although the performance of current blockchains is much lower than that of a traditional database [2], the performance of permissioned blockchains is often better than that of permissionless ones. Meanwhile, permissioned blockchains are more suitable for building enterprise use cases, in which authentication is required but participants do not trust each other. Besides, permissioned blockchains have many advantages compared with permissionless ones, such as authority management, multiple channels, and data privacy protection. Currently, Hyperledger Fabric (Fabric) [3], with IBM as the main maintainer, is the most popular permissioned blockchain platform, and has already been applied to various industries, such as supply chain [4], healthcare [5], and smart grids [6].

The performance of Hyperledger Fabric is one of the biggest factors that restricts the development of enterprise blockchain applications, especially for the high-throughput financial scenarios, such as stock exchanges, credit card companies, and mobile payment platforms. As we all know, the Proof of Work (POW) consensus is the main performance bottleneck of the permissionless blockchains such as Bitcoin [7] and Ethereum [8]. Unlike permissionless ones, many recent studies [9, 10] have highlighted the V&C module as the main bottleneck of Fabric, especially after Raft [11] is used as the consensus in Fabric v1.4. Although some optimizations [12, 13] have been proposed to optimize the V&C module, the improvement effect is not very significant. Some have limitations of application scenarios. In this paper, we focused on improving the performance of Fabric.

First, we divided the V&C module into five main sub-steps and conducted a full performance evaluation for them. Then according to the analysis of the evaluation reports and our understanding of the V&C

URI: https://hdl.handle.net/10125/80232
978-0-9981331-5-7

module, we re-structured it by introducing pipelined execution on the five sub-steps, and also proposed two pipeline acceleration schemes for the pipelines with lower processing speed. These optimizations helped process multiple blocks in parallel and reduce the average processing time of blocks.

We implemented these optimizations in Hyperledger Fabric v2.1. Based on our experiments, we found that the throughput was improved by 4.38× when LevelDB was used as the state database (StateDB), and the performance was improved by at least 2× with CouchDB, even achieving 4.42× with a small *block size* (the number of transactions contained in a block) of 20. Most importantly, our proposed optimizations followed the original transaction processing flow and were transparent to blockchain applications.

The rest of this paper is structured as follows: Section 2 introduces the latest studies. Section 3 provides a detailed introduction to the transaction flow of Fabric. Section 4 describes our evaluation method in detail and the fine-grained analysis of the V&C module, while Section 5 presents our proposed architecture against the vanilla Fabric. Section 6 gives a detailed performance analysis of our optimized Fabric. Section 7 summarizes all our work.

2. Related work

The performance of Hyperledger Fabric is not acceptable in some scenarios [14, 15], which restricts the development of enterprise blockchain technologies to some extent. Therefore, improving the performance of Hyperledger Fabric, especially the throughput of transaction, has attracted great attention from both research and industry communities. We reviewed the latest performance studies on Hyperledger Fabric here.

Dang et al. [16] designed an efficient sharding protocol and a general distributed transaction protocol to improve transaction throughput at scale. Nguyen et al. [17] presented a comprehensive performance analysis of Fabric v1.1 on a realistic set-up of up to 48 cluster nodes using a production-ready Kafka ordering service. Their results showed that the main scalability bottleneck was the number of endorsing peers, because scaling the Kafka cluster did not affect the overall performance. This conclusion was not rigorous because whether increasing the number of endorsement peers or reducing it would not affect the block processing speed of peers. Furthermore, the ordering service has been migrated from Kafka to Raft since Fabric v1.4. Sharma et al. [18, 19] proposed to reorder transactions to reduce transaction conflicts in the ordering service, in a bid to reduce the transaction failure rate. We believe that these techniques are suitable for high-conflict application scenarios, but will not have much effect on general scenarios.

Gorenflo et al. [12] proposed a basket of optimizations to achieve a throughput of 20,000 transactions/second. However, many of the proposed optimizations were too ideal. For example, all the blocks and states were stored in memory to achieve the results of having no disk IO and the V&C module and endorser were split into different servers, and they also assumed the transactions without reading-writing conflicts. These optimizations would increase the actual burden on deployment and were obviously impossible in practical terms.

Thakkar et al. [13, 20] found that the main bottleneck of Fabric was the step of validation system chaincode (*VSCC*) through a comprehensive performance evaluation, and provided some guidelines about how to write smart contracts and deploy Fabric networks to achieve high performance. Also, they implemented some optimizations such as validating transactions in parallel based on the analyzed cross-block transaction dependency graph. We thought this optimization would only work in high-conflict scenarios. Based on these studies, Javaid et al. [9] proposed to cache chaincode information to reduce the number of StateDB accesses and parallel some commit operations. Some of these optimizations have already been integrated into Hyperledger Fabric, and Dreyer et al. [21] proved the effectiveness of these optimizations on Fabric v2.0.

In summary, although some of the proposed optimizations have been accepted by the new versions of Fabric, the V&C module remains the performance bottleneck [9, 20]. At that, we provided a pipelined execution scheme to re-structure the V&C module to further improve Fabric's performance on Fabric v2.1 while ensuring its universality.

3. Transaction flow of Fabric

Client, endorsing peer, and ordering node are the three kinds of entities in a Fabric network. Each entity should be authenticated by a Membership Service Provider (MSP) [22] before joining this network, and communicates with each other through gRPC protocol. The business logic is written by smart contracts (chaincodes) which can be implemented in any programming language [23]. Before committing a transaction into the ledger, the three kinds of entities should take four phases to process it (as shown in Figure 1). Since our optimizations were focused on some of these phases, we gave a brief introduction to them here.

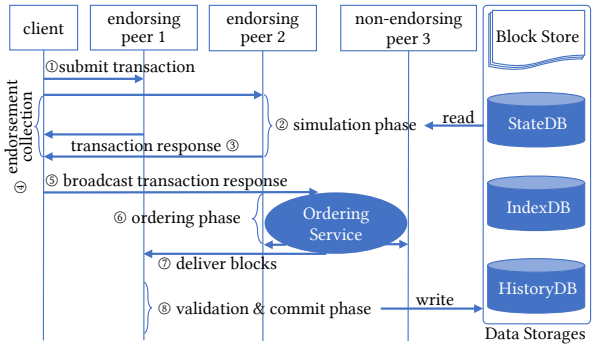


Figure 1. Transaction flow in Hyperledger Fabric

A. Simulation phase. A client sends a transaction to a group of endorsing peers that meet the pre-set endorsement policy [24], and all the selected peers process this transaction independently. Each endorsing peer first judges whether the transaction is compliant, and then takes the transaction proposal inputs as the arguments of the invoked function of the chaincode. Then, the chaincode executes chaincode apis against the StateDB to produce transaction results. Please note that no updates are made to the ledger at this point. If there are no errors during the simulation, a "transaction response" with endorsement results will be delivered back to the client.

B. Ordering phase. A transaction will be broadcasted to the ordering service once enough endorsement results have been collected by the client. The ordering service which composed of ordering nodes adopts a pre-set consensus protocol to sort all the received transactions. It is noted that the Raft consensus was recommended in Fabric v1.4.1, and both the Kafka consensus and the Solo consensus were deprecated in Fabric v2.x. Once the pre-set number of transactions or time interval is reached, the ordering service will pack the ordered transactions into a block and deliver it to all the peers after it is signed.

C. Validation phase. Once a block delivered from the ordering service is received, a peer will first check the effectiveness of the block structure and endorsement policy of transactions through the *VSCC* validation. Then the version conflict of all the reading states of each transaction will be checked by the multi-version concurrency control (*MVCC*) validation. This step is used to prevent the double-spending problem [25]. In the above checks, as long as one item is not satisfied, a transaction will be marked as invalid.

D. Commit phase. A block along with the information within it will be updated to the four types of data storages [26] after the validation phase. First, the block along with its validation information will be added into the *Block Store* which is stored in a file system and organized like a chain. Then, the position and offsets of this block and the transactions within it will be recorded into the *IndexDB* for quick queries. After that, all the writing states along with their new version numbers in the valid transactions will be updated into the *StateDB*. At last, the changelog of each updated state will be recorded into the *HistoryDB* for supporting the historical query of states.

4. Evaluation methodology

4.1. Experimental situation

A. Fabric network set-up. The Fabric network, which was established for all experimental analysis and performance evaluations, consists of four endorsing peers, which were divided into two organizations. In this network, the ordering service was deployed on three nodes. Specifically, the Raft was applied as the consensus protocol, one channel containing all the peers was established, and a transaction with at least one endorsement result could be broadcasted to the ordering service. Here, seven clients were used to submit transaction proposals where each client runs eight threads. All endorsing peers, ordering nodes, and application clients were running on separate physical machines. Each endorsing peer was allocated with an Intel Xeon E5-2670 v2 @ 2.5 GHz (10 cores with 20 threads), 64 GB RAM clocked at 1600 MHz, and 1 T HDD. Each ordering node or client was allocated an i5-9400 @ 2.9 GHz (6 cores with 6 threads), 8 GB RAM clocked at 2666 MHz, and 1 T HDD. All the machines were connected through a 1 Gbps network. Although our network contained only two organizations, our experimental results were also applicable to larger networks for the abilities of transaction simulation and block transmission can be scaled up by adding peers, ordering nodes, and bandwidth [20].

Table 1. StateDB operation times of smallbank

Functions	Reading times	Writing times
Account Creation	0	1
Transfer Money	2	2
Deposit Cash	1	1
Account Destroy	0	1

B. Application. The typical financial application *smallbank* which was peeled from Hyperledger Caliper [27] was used to evaluate our network. Its smart contract implemented four functions and each function warranted a certain amount of reading and writing operations on the StateDB. Table 1 listed the number of times of the functions to read and write the StateDB.

For each experiment, all clients sent a total of 50,000 transactions to invoke these functions randomly. The total transaction sending rate was close to the saturation point [20] of the processing capacity of the network.

C. Metrics. We examined the V&C module critically for it's still the performance bottleneck of Fabric. The metric *throughput* was defined to express the rate of committing transactions into the ledger, and several *latencies* were used to express the processing speed of the following sub-steps of the V&C module:

- *vscc:* The latency of verifying a block's structure and endorsement policies of transactions within it.
- *mvcc:* The latency of verifying the double-spending problem for all transactions.
- *ledger_write:* The latency of committing a block to the *Block Store* and adding position indices of this block and its transactions to the *IndexDB*.
- *statedb_write:* The latency of updating the states from valid transactions to the *StateDB*.
- *historydb_write:* The latency of recording the changelogs of states to the *HistoryDB*.
- *others:* The latency of trivial operations.

We used the performance traffic engine (PTE) [28] instead of Caliper [27] as clients to submit transaction proposals for Caliper did not support Fabric v2.x when we performed experiments. For PTE and Capliper were both prone to missing block events and can't log the above latencies [10], we inserted various record points through the source code of the V&C module to record these latencies. All experiments were repeated five times and took the mean values as the metrics.

4.2. Analysis of the V&C module

For the V&C module is still the main bottleneck of Fabric [9, 10]. Therefore, we focused on studying the latency of the steps of the V&C module. Currently, Fabric supports two types of key-value StateDB, they are LevelDB and CouchDB. LevelDB is the default embedded StateDB, it provides faster access on keys, but its query function is simple. CouchDB is an external StateDB and accessed via Rest Api. Although its access speed is slower than LevelDB, it provides rich query functions like range query and fuzzy query to support more complex business logic. Using different StateDBs has different effects on performance, so we evaluated the latency of the V&C module in different types of StateDB and different block sizes. All our evaluations were performed on Fabric v2.1.

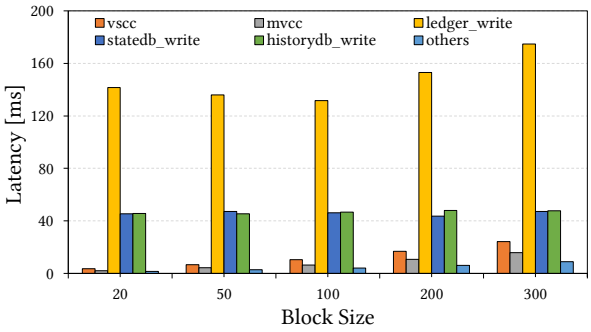


Figure 2. Latency evaluation of the original V&C module with varying block sizes under LevelDB

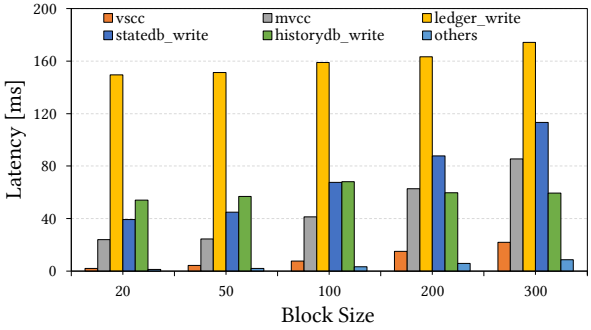


Figure 3. Latency evaluation of the original V&C module with varying block sizes under CouchDB

1) LevelDB as StateDB: Figure 2 showed the details of the V&C latencies with different block sizes for LevelDB. It's worth noting that:

- The *ledger_write* was the lowest step. It almost accounted for half of the total latency. But the latency growth was much lower than the linear growth of block size.
- The latencies of *vscc* and *mvcc* increased according to the block size, but only occupied a small part of the total latency.
- For the LevelDB provided relatively fast accesses, the latencies of *statedb_write* and *historydb_write* were very close to each other and only increased slightly with block size increases.

2) CouchDB as StateDB: Figure 3 showed the evaluation results for CouchDB. The latency distribution was some different with that of LevelDB. We listed our observations below:

- The *ledger_write* remained the main bottleneck. This is because there is no need to operate the StateDB when we write the ledger. The latency

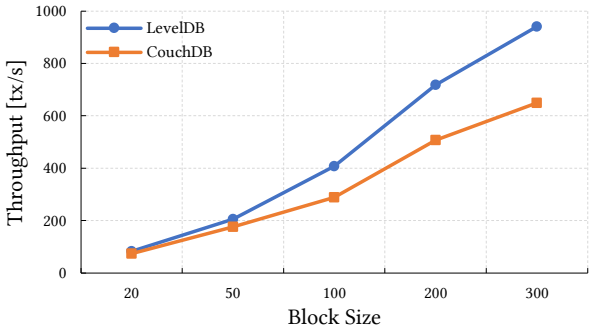


Figure 4. Performance comparison of the vanilla Fabric with different StateDBs

under a certain block size was almost the same as that of LevelDB.

- The *mvcc* latency was almost 6× slower than that of LevelDB. This is owing to accessing CouchDB for states through Rest Api is slower than LevelDB. Likewise, the latency of *statedb_write* was also significantly larger.

- The latency of *vscc* was not significant and was more or less the same as that in LevelDB. This is because a cache is employed to store the chaincode information after the first access to the StateDB since Fabric v1.4.

Figure 4 showed that the throughput increased as the block size increased. This is because the increase in block size is higher than that of the total latency of processing a block. Under saturation-point testing [20], the relationship among throughput, total latency, and block size satisfied formula $throughput = 1000/total\_latency \times block\_size$. Here, we can conclude that the cost of the V&C module was more friendly for larger blocks. We also observed that the throughput was reduced under any block size for the total latency increased when we used slower CouchDB.

5. Optimization

The existing version of Fabric has implemented many parallel operations in the V&C module to improve CPU utilization, such as parallel verification in the *vscc* step and parallel database reading in the *mvcc* step. However, the CPU can only process one step of the V&C module for a block at any time, which limits the utilization rate of the CPU to some extent. We proposed to introduce pipelined execution for the V&C module to process several blocks parallel, with an aim to achieve the purpose of increasing CPU usage and throughput.

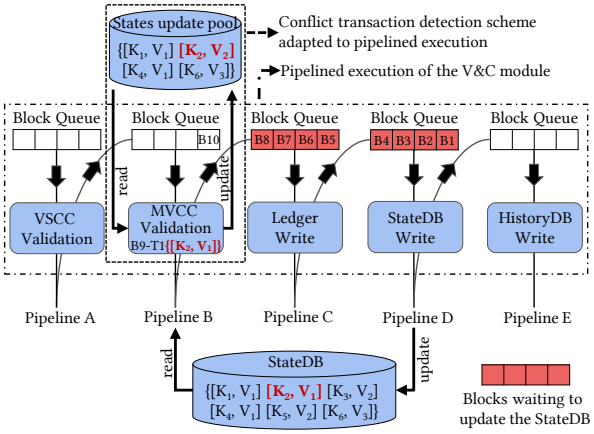


Figure 5. Pipelined execution of the V&C module

5.1. Pipelined execution

Pipelined processing [29] is a type of parallelism. It can simultaneously process multiple different operations while ensuring the processing order of all operations of a data stream. The simultaneous execution of multiple operations of different data streams can improve CPU usage and throughput.

As the metrics described in Section 4.1-C, the V&C module of Fabric was divided into five main steps: *vscc*, *mvcc*, *ledger_write*, *statedb_write*, and *historydb_write*, in addition to the system overhead *others*. In the V&C module, the blocks must be strictly executed according to the order received from the ordering service. The processing steps of each block also cannot be changed. The execution of a subsequent step only depends on the previous step. Therefore, the V&C module is naturally suitable for pipeline transformation.

We re-structured the V&C module using pipelining, and Figure 5 demonstrated our designed constructure. Each pipeline was responsible for processing a step of the V&C module and configured with a block queue to receive blocks delivered by the previous one. The received blocks will be processed in sequence and then delivered to the next pipeline after processing. In this way, the V&C module can process several blocks, while ensuring the execution sequence of blocks and the five steps of the V&C module.

It is noteworthy that the original *mvcc* validation was not suitable for our optimized V&C module any more. In the vanilla Fabric, the *mvcc* compared a state's version with the StateDB and the states of its previous transactions of the same block to check for conflict. After pipelining, the pipelines and their block queues may contain blocks that have not been processed in a timely manner. Especially, the pipelines that had

slower processing speeds, such as *ledger_write* and *statedb_write*. Therefore, the state versions obtained by the *mvcc* pipeline in the original way were likely not the latest versions, because the pipeline *ledger_write*, *statedb_write* and their block queues may contain the states with latest version numbers but haven't been updated to the StateDB yet.

As a result, we created a new state update pool, as shown in the vertical dashed box in Figure 5, to store the write-states of each valid transaction that was verified by *mvcc* temporarily. The state update pool covered all the write-states which were waiting to be updated to the StateDB in pipelines *ledger_write* and *statedb_write*. After a block was updated to the StateDB by pipeline *statedb_write*, the write-states related to this block will be deleted from this pool. A transaction marked as valid must meet two conditions: *1)* The version number of the states in the reading set of the transaction should be consistent with the states in StateDB. *2)* The states in the reading set of the transaction should not exist in this state update pool.

5.2. Pipeline acceleration schemes

After pipeline transformation, the average processing time of the V&C module for a block is theoretically equal to that of the slowest pipeline, which can greatly increase the processing efficiency compared with the original way of sequential execution. Therefore, it's a natural way to further improve the processing efficiency by increasing the processing speed of the otherwise slower pipelines.

Figure 2 showed that when LevelDB was employed as the StateDB, *ledger_write* was the main performance bottleneck. In Figure 3, in addition to *ledger_write*, *statedb_write* also gradually became a bottleneck when the block size increased. This is because it takes more time to access external CouchDB through Rest API than to access LevelDB. However, we found that the growth rate of processing time of *ledger_write* and *statedb_write* was much slower than that of the block size. Therefore, we used this discovery to accelerate the processing of the *ledger_write* and *statedb_write* pipelines.

Since the pipelines of slower processing speed cannot keep up with the pipelines of faster processing speed, it was easier to accumulate unprocessed blocks in the pipelines of slower processing speed. Based on the characteristics revealed in our experimental data, we can process multiple blocks (up to 10 blocks) at one time in the pipelines of *ledger_write* and *statedb_write*, so as to decrease the number of disk writing and the number of accessing StateDB to speed up the processing speed of these two pipelines. Figure 6 depicted our pipeline

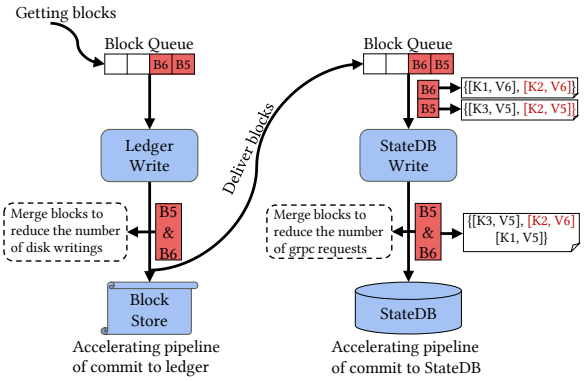


Figure 6. Pipeline acceleration schemes

acceleration schemes. For the *ledger_write* pipeline, we merged multiple blocks for one-time writing to reduce the disk writing overhead. It should be noted that the calculation method of position offset of each block and the transactions within it should be improved here. We must consider the total length of the blocks before it. After multiple blocks are written at one time, the ledger save-point should be updated by using the information of the highest block height of the merged blocks. For the *statedb_write* pipeline, we identified the writing states from multiple blocks and performed a one-time writing operation. Once different blocks contain a same state, the state with the highest block height will be selected to write.

Furthermore, we calculated the average processing latency of a block in each pipeline using the recorded logs at various points through the V&C module, and then used *max_pipeline_latency* to represent the maximum average latency of these five pipelines. Under saturation-point testing, this was approximately equal to the average processing time of a block. Therefore, the throughput after pipelining can be deduced by the formula $throughput = 1000/max\_pipeline\_latency \times block\_size$.

6. Experimental results

6.1. Implementation

The optimizations we proposed in Section 5 were implemented on the V&C module of Fabric v2.1. Five pipelines as shown in Figure 5 were set to run the six components described in Section 4.1-C. The *mvcc*, *ledger_write*, *statedb_write*, and *historydb_write* were separated from the original V&C module and executed with one pipeline respectively. Then, the remaining two components *vscc* and *others* were executed in the *vscc* pipeline. All the parallel pipelines were implemented as go-routines, and the first-in-first-out principle was

adopted for the block queues to ensure the processing order of blocks. The length of the block queues was uniformly set to 200, which was the same as that of the block queue used to receive blocks from the ordering service. Whenever the block queue of a pipeline received a block, it would notify the execution module of this pipeline to perform the corresponding validation or commit operation through a message mechanism. A pipeline will sleep for 100 ms once the length of the next pipeline's block queue reaches the upper limit. When the pipelines of *ledger_write* and *statedb_write* processed multiple blocks at one time, a temporary array was used to save the multiple blocks popped from the queue, and then the blocks were merged and written into the Block Store or the StateDB. After this operation was completed, the blocks in the temporary array were sent to the next pipeline in turn.

6.2. Evaluation of the optimized Fabric

The performance of our optimized Fabric was evaluated in this section. The set-up, application, and the number of experiments used for the study were the same with the description in Section 4.1.

1) LevelDB as StateDB: The results for LevelDB with different block sizes were reported in Figures 7 and 8. We compared the latency of *vscc* and *mvcc* with the latency of the vanilla Hyperledger Fabric in Figure 2, and found that the latency of *vscc* was almost unchanged, but the latencies of *mvcc* and *historydb_write* increased slightly after pipelining under any block size. This is because the five V&C steps in the vanilla Fabric are processed sequentially, and there was no competition for computing resources in these steps. However, after pipeline transformation, all pipelines were executed in parallel, and there was a problem of competition for computing resources among the pipelines, so the processing time of the corresponding step has been extended. Although LevelDB provided faster access speed, under the premise of competing for computing resources, the latency was also increased. We took block size of 200 as an example. The *mvcc* latency was 1.78× the original latency (from 10.7 to 19 ms), and the *historydb_write* latency was 1.21× the original latency (from 47.9 to 58.1 ms).

The latency changes of *ledger_write* and *statedb_write* were surprising: taking a block size of 200 as an example, the latency of *ledger_write* was reduced from 153.2 ms to 41.1 ms (a 3.7× improvement), and the latency of *statedb_write* was reduced from 43.6 ms to 9.2 ms (a 4.7× improvement). The processing speed increased in *ledger_write* and *statedb_write* steps was entirely due to the implementation of the

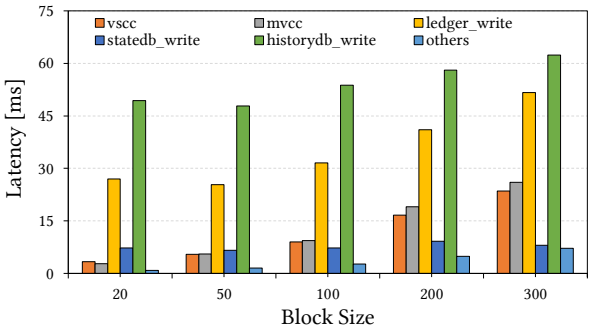


Figure 7. Latency evaluation of the optimized V&C module with varying block sizes under LevelDB

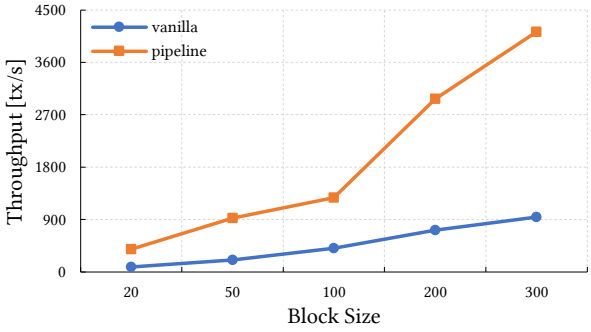


Figure 8. Performance comparison with the vanilla Fabric: LevelDB vs. Block Size

pipeline acceleration schemes. This showed that our previous analysis in Section 5.2 was correct. Reducing the number of disk writing and database accesses can reduce the processing latencies of these two steps. We counted the number of blocks written to the Block Store and committed to the StateDB each time in Table 2, and found that the number of blocks processed by pipeline *ledger_write* and *statedb_write* was the same under different block sizes. This is because when LevelDB is used as StateDB, *ledger_write* is the only performance bottleneck, as shown in Figure 2. Therefore, this pipeline was most likely to accumulate blocks. When the pipeline *ledger_write* processed multiple blocks at one time and threw the blocks to pipeline *statedb_write*, the thrown blocks would also be processed at one time for the processing speed of pipeline *statedb_write* was more faster.

Table 2. Average number of blocks processed each time: LevelDB vs. Block Size

Pipeline \ Block size	20	50	100	200	300
ledger_write	8.1	8.2	8	7.8	7.6
statedb_write	8.1	8.2	8	7.8	7.6

Figure 8 indicated the throughput was more higher than that of the vanilla Fabric. We achieved up to a 4.38× improvement in throughput (from 947 to 4,125 transactions/second) and a 27% increment in CPU utilization (from 22% to 49%) with a block size of 300.

2) CouchDB as StateDB: As shown in Figure 3, when CouchDB was used as the StateDB, the *ledger_write* was a relative time-consuming task, and the *statedb_write* gradually became another time-consuming task as the block size increased. After the pipelined execution was completed, Figure 9 indicated that the latencies of *vscc*, *mvcc*, and *historydb_write* were increased of any block size, which was like LevelDB. However, the latency growth of these three steps was greater than that of LevelDB. We took a block size of 200 as an example. The latency of *vscc* was increased from 14.9 ms to 34.8 ms (about 2.3×). The *mvcc* latency increased significantly by about 2.3×, from 62.8 ms in Figure 3 to 146.3 ms. The reason is the same as we discussed in terms of LevelDB: the concurrent execution among pipelines intensifies the competition for the available CPU resources. Meanwhile, the slower access speed of CouchDB makes the competition more intense, which is more obvious for *mvcc* as it requires concurrent access to CouchDB, but the *history_write* pipeline is not a computationally intensive task, so its latency only increases a little.

Comparing Figure 3 with Figure 9, we found that the average block processing time of *ledger_write* was greatly reduced especially the block size was small. But the processing time of *statedb_write* did not decrease, but increased a little bit. However, this was understandable because both *mvcc* and *statedb_write* needed to access the slower CouchDB concurrently, which further intensified competition for computing resources. If the pipeline acceleration schemes were not used here, the latency of *ledger_write* and *statedb_write* could be higher. Benefits from the proposed optimizations, the throughput was still greatly improved. The Figure 10 showed the throughput results. It achieved at least 2× under any block size, about 4.3× improvement for small block size of 20 and 50, and got a maximum throughput of 1,309 transactions/second for a block size of 300. For CPU utilization, it achieves a 16% increment for a block size of 300 (from 21% to 37%).

Another interesting finding in Figure 9 was that when the block size increased, the processing speed of the *mvcc* pipeline gradually decreased until it became the slowest pipeline. This will cause a large number of blocks to accumulate in its block queue, so the pipeline *mvcc* cannot deliver enough blocks to the pipeline *ledger_write* timeously, resulting in a gradual decrease in the number of blocks processed by pipeline

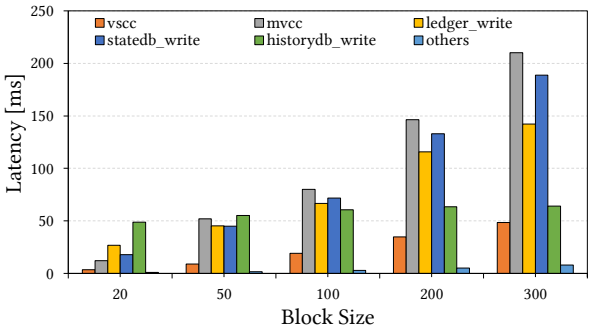


Figure 9. Latency evaluation of the optimized V&C module with varying block sizes under CouchDB

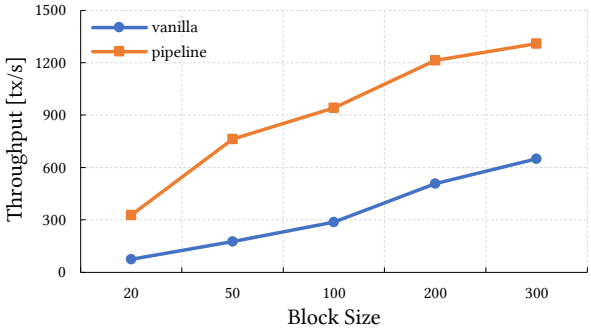


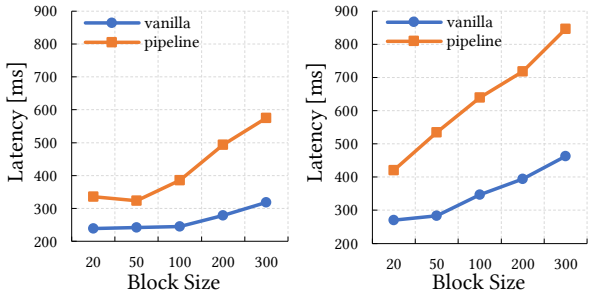
Figure 10. Performance comparison with the vanilla Fabric: CouchDB vs. Block Size

ledger_write and *statedb_write* at one time (Table 3).

Table 3. Average number of blocks processed each time: CouchDB vs. Block Size

Pipeline \ Block size	20	50	100	200	300
ledger_write	8	4.48	3.25	1.73	1.34
statedb_write	8	4.8	3.62	2	1.72

3) Latency analysis: Although our optimized scheme got a better performance improvement, it inevitably caused some minor negative impacts on transaction latency. From Figure 11 we can see that the total latency became longer whether LevelDB or CouchDB was used. It should be noted that the total latency of the V&C module was equal to the sum of the latencies of all steps, and it became longer mainly due to the computing resources competition of the pipelines. Another reason caused the longer latency was the pipelines *ledger_write* and *statedb_write* process multiple blocks at one time. Although the average processing time of each block became shorter, the latencies in these two pipelines were equal to the average processing time of blocks multiplied by the average number of blocks. We took a block size of 300 under LevelDB for example. The average number of



(a) Using LevelDB (b) Using CouchDB

Figure 11. Total latency comparison of the V&C module with the vanilla Fabric

processed blocks in *ledger_write* pipeline was 7.6 per time (Table 2), and the average latency of processing a block was 51.7 ms (Figure 7), so the average latency of this pipeline was $51.7 \times 7.6 = 392.92ms$. Similarly, this calculation method was also applicable to pipeline *statedb_write* operations. However, the maximum latency with the saturation-point testing was about 850 ms when the block size was set to 300 under CouchDB, which was still acceptable in most actual production environments.

7. Conclusion

We performed a comprehensive evaluation of the V&C module of Fabric, which was considered as the main bottleneck. Hence, some optimizations were proposed to optimize this bottleneck based on the detailed analysis. We divided the V&C module into five independent steps, and then applied pipelined execution and pipeline acceleration schemes for them to process multiple blocks simultaneously while ensuring the processing sequence thereof. We implemented our optimizations in Fabric v2.1, and the results showed that the throughput improved by 4.38× for LevelDB (from 947 to 4,125 transactions/second), while the improvement with CouchDB was 4.42× with a small block size of 20, and about 2× with a block size of 300 (from 649 to 1,309 transactions/second).

Furthermore, all the techniques described here were different from the parallel execution proposed in previous works, and were very suitable to be integrated into a new version of Fabric. Furthermore, these techniques were transparent to applications. As a result, our optimizations can improve the performance of any Fabric-based application.

References

[1] C. V. Helliar, L. Crawford, L. Rocca, C. Teodori, and M. Veneziani, "Permissionless and permissioned blockchain diffusion," *International Journal of Information Management*, vol. 54, p. 102136, 2020.

[2] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*, (New York, NY, USA), pp. 1085–1100, ACM, May 2017.

[3] E. Androulaki, A. Barger, V. Bortnikov, and et al., "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, no. 30 in EuroSys '18, (New York, NY, USA), p. 15, ACM, 2018.

[4] Q. Song, Y. Chen, Y. Zhone, K. Lan, and S. Fong, "A supply-chain system framework based on internet of things using blockchain technology," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 1, p. 24, 2021.

[5] S. Tanwar, K. Parekh, and R. Evans, "Blockchain-based electronic healthcare record system for healthcare 4.0 applications," *Journal of Information Security and Applications*, vol. 50, p. 102407, 2020.

[6] A. Lohachab, S. Garg, B. H. Kang, and M. B. Amin, "Performance evaluation of hyperledger fabric-enabled framework for pervasive peer-to-peer energy trading in smart cyber-physical systems," *Future Generation Computer Systems*, vol. 118, pp. 392–416, 2021.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," tech. rep., Manubot, 2008.

[8] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform." `https://ethereum.org/en/whitepaper/`.

[9] H. Javaid, C. Hu, and G. Brebner, "Optimizing validation phase of hyperledger fabric," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 269–275, IEEE, 2019.

[10] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance characterization of hyperledger fabric," in *2018 Crypto Valley conference on blockchain technology (CVCBT)*, (Zug), pp. 65–74, IEEE, June 2018.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 305–319, USENIX Association, USENIX ATC'14, 2014.

[12] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 455–463, IEEE, May 2019.

[13] P. Thakkar and S. Natha, "Scaling hyperledger fabric using pipelined execution and sparse peers," *arXiv preprint arXiv:2003.05113*, 2020.

[14] T. Nakaike, Q. Zhang, Y. Ueda, T. Inagaki, and M. Ohara, "Hyperledger fabric performance characterization and optimization using goleveldb benchmark," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, p. 9, IEEE, May 2020.

[15] C. Wang and X. Chu, "Performance characterization and bottleneck analysis of hyperledger fabric," *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, Nov 2020.

[16] H. Dang, T. T. A. Dinh, D. L.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data (SIGMOD'19)*, (New York, USA), pp. 123–140, ACM, June 2019.

[17] M. Q. Nguyen, D. Loghin, and T. T. A. Dinh, "Understanding the scalability of hyperledger fabric," *BCDL VLDB*, 2019.

[18] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "How to databasify a blockchain: the case of hyperledger fabric," *arXiv preprint arXiv:1810.13177*, 2018.

[19] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 105–122, 2019.

[20] P. Thakkar, S. N. N, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 264–276, IEEE, Sep 2018.

[21] J. Dreyer, M. Fischer, and R. Tönjes, "Performance analysis of hyperledger fabric 2.0 blockchain platform," in *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, pp. 32–38, IEEE, November 2020.

[22] "Membership service providers (msp)." `https://hyperledger-fabric.readthedocs.io/en/latest/msp.html`.

[23] "Smart contracts and chaincode." `https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html`.

[24] M. Yacov, B. Artem, and T. Yoav, "Endorsement in hyperledger fabric via service discovery," *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 2–1, 2019.

[25] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun, "Misbehavior in bitcoin: A study of double-spending and accountability," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, p. 32, 2015.

[26] E. Zhou, H. Sun, B. Pi, and et al., "Ledgerdata refiner: A powerful ledger data query platform for hyperledger fabric," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, (Granada, Spain), pp. 433–440, IEEE, Oct 2019.

[27] "Hyperledger caliper: A blockchain performance benchmark framework for hyperledger fabric." `https://github.com/hyperledger/caliper`.

[28] "Performance traffic engine - pte." `https://github.com/hyperledger/fabric-test/tree/master/tools/PTE`.

[29] "Pipeline processing." `https://en.wikipedia.org/w/index.php?title=Pipeline_(computing)&oldid=999661022`.